Introduction

Table of Contents

Computer system

A *computer system* consists of hardware and systems software that work together to run application programs.

Structure of "Hello World" program in C

```
#include <stdio.h>

int main(){
  printf("hello, World\n");
}
```

- The *hello* program begins life as *source program* (or a *source file*) that the programmer creates with an editor and saves in a text file called `hello.c`.
- The source program is a sequence of bits, each with a value of 0 or 1, organized in 8-bit chunks called *bytes*.
  - Each byte represents some text character in the program
- Most modern systems represent text characters using the **ASCII** standard that represents each character with a unique byte-sized integer value.

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46

h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123

\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108

l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

- The `hello.c` program is stored in a file as a sequence of bytes.
  - Each byte has an integer value that corresponds to some character.
  - For example —
    - The first byte has the integer value 35, which corresponds to the character '#'.
    - The second byte has the integer value 105, which corresponds to the character 'i'.
  - Each text line is terminated by the invisible *newline character* '\n', which is represented by the integer value 10.
- Files such as `hello.c` that consist exclusively of ASCII characters are known as *text files*. All other files are known as *binary files*.

All information in a system — including disk files, programs stored in memory, user data stored in memory, and data transferred across a network is represented as bunch of bits.

The only thing that distinguishes different data objects is the context in which they are viewed.

- The same sequence of bytes might represent an integer, floating-point number, character string, or machine instruction

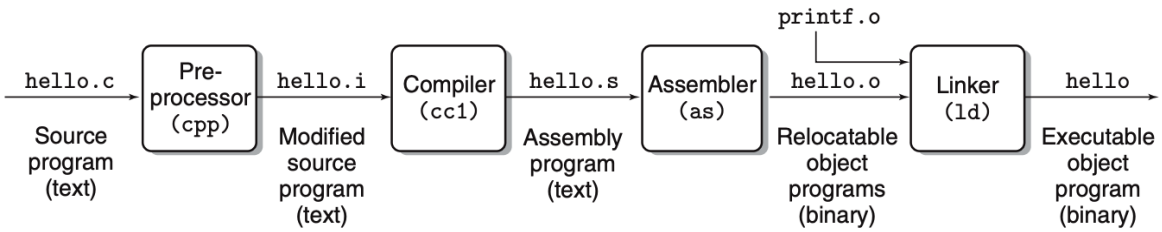
Translation of "hello.c" to executable object program (hello)

The "Hello World" program begins life as a high-level C program because it can be read and understood by human beings in that form.

- However, in order to run `hello.c` on the system, the individual C statements must be translated by other systems into a sequence of low-level *machine-language instructions*.
- These instructions are then packaged in a form called an *executable object program* and stored as a binary disk file.
- On a unix system, the translation from source file to object file is performed by a *compiler driver*:

  ```
  unix> gcc -o hello hello.c
  ```

- The *GCC* compiler driver reads the source file **hello.c** and translates it into an executable object file `hello`.
- The translation is performed in sequence of four phases.
  - The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are collectively known as *compilation system*.



Preprocessing Phase

- The preprocessor (cpp) modifies the original C program according to directive that begin with the # character.
    - For example — The `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system head file `stdio.h` and insert it directly into the program text.
- The result is another C program, typically with the *.i* suffix.

Compilation Phase

- The compiler (cc1) translates the text file `hello.i` into the text file `hello.s` which contains *assembly-language program*
- Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form

Assembly Phase

- The assembler (as) translates `hello.s` into *machine-language instructions*.
- Then packages them in a form known as *relocatable object program* and stores the result in the object file `hello.o`.
- The `hello.o` file is a binary file whose bytes encode machine language instructions rather than characters.

Linking Phase

- The **hello** program calls the `printf` function, which is a part of the *standard C library* provided by every C compiler.
- The `printf` function resides in a separate precompiled object file called `printf.o`, which is to be merged with the **hello.o** program.
- The linker (ld) handles this merging.
- The result is `hello` file, which is an *executable object file* that is ready to be loader into memory and executed by the system.

Executing "Hello" Program

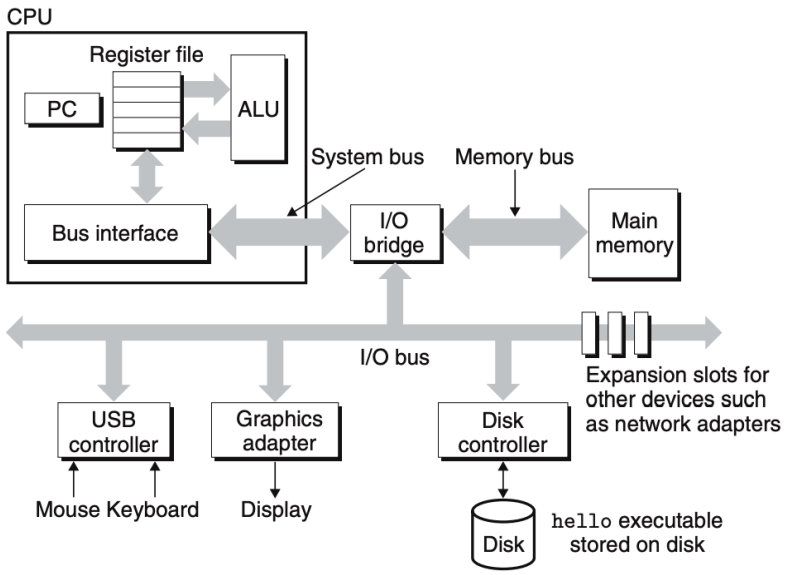
At this point, `hello.c` source program has been translated by the compilation system into an executable object file called `hello` that is stored on disk.

- To run the executable file on a Unix system, type its name to an application known as a *shell*

```
unix> ./hello
hello, world
unix>
```

- The **hello** program prints its message to the screen and then terminates.

Hardware Organization of a system



This particular picture is modeled after the family of Intel Pentium systems, but all systems have a similar look and feel.

Buses

Running throughout the system is a collection of electrical conduits called *buses* that carry bytes of information back and forth between the components.

- Buses are typically designed to transfer fixed-sized chunks of bytes known as *words*.
- The number of bytes in word (the *word size*) is a fundamental system parameter that varies across systems.
  - Most machine today have word sizes of either 4 bytes (32 bits) or 8 bytes (64 bits).

I/O Devies

Input/Output (I/O) devices are the system's connection to the external world.

- Example — Keyboard and mouse for user input, a display for user output, and a disk drive for long-term storage of data and programs.
- Initially the **hello** program resides on the disk.

Each I/O device is connected to the I/O bus by either a *controller* or an *adapter*.

- Controllers are chip sets in the device itself or on the system's main printed circuit board (*motherboard*).
- An adapter is a card that plugs into a slot on the motherboard.

Main Memory

The *main memory* is a temporary storage device that holds both a program and the data it manipulates while the processor is executing the program.

- Physically, the main memory consists of a collection of *dynamic random access memory* (DRAM) chips.
- Logically, memory is organized as a linear array of bytes, each with its own unique address (array index) starting at 0.
- The size of data items that correspond to C program variables vary according to type.
    - For example — on an IA32 machine running Linux, data types `int`, `float`, and `long` require 4 bytes, and type `double` require 8 bytes.

Processor

The *Central Processing Unit* (CPU), is the engine that interprets (or *executes*) instructions stored in main memory.

- At its core is a word-sized storage device (or *register*) called the *program counter* (PC). At any point in time, the PC points at (contains the address of) some machine-language instruction in main memory.

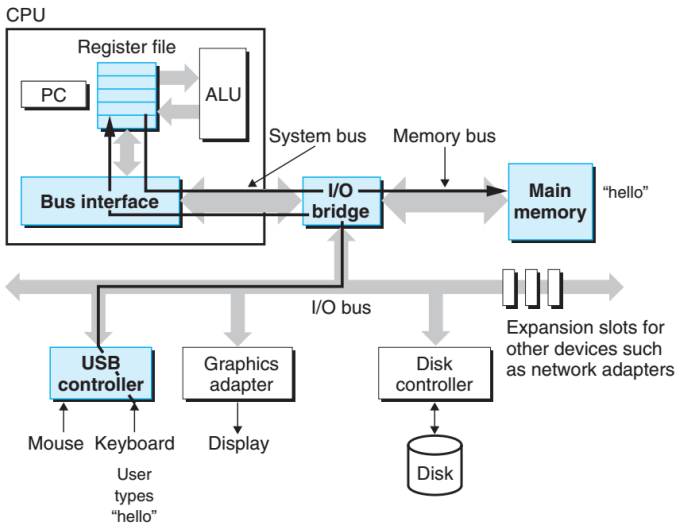
A process appears to operate according to a very simple instruction execution model, defined by its *instruction set architecture*.

- There are only a few of these simple operations, and they revolve around main memory, the register file, and the *arithmetic/logic unit* (ALU).

    - The register file is a small storage device that consists of a collection of word-sized registers, each with its own unique name.
    - The ALU computes new data and address values.
- Example of simple operations that CPU might carry out:
    - *Load:* Copy a byte or a word from main memory into a register, overwriting the previous contents of the register
    - *Store:* Copy a byte or a word from a register to a location in main memory, overwriting the previous contents of that location.
    - *Operate:* Copy the contents of two registers to the ALU, perform an arithmetic operation on the two words, and store the result in a register, overwriting the previous contents of that register.
    - *Jump:* Extract a word from the instruction itself and copy that word into the program counter (PC), overwriting value of the PC.

Mechanics of Running the "hello" program

- Initially the shell program is executing its instructions, waiting for the command.
- Once the command is typed `./hello`, the shell program reads each one into a register, and then stores it in memory.

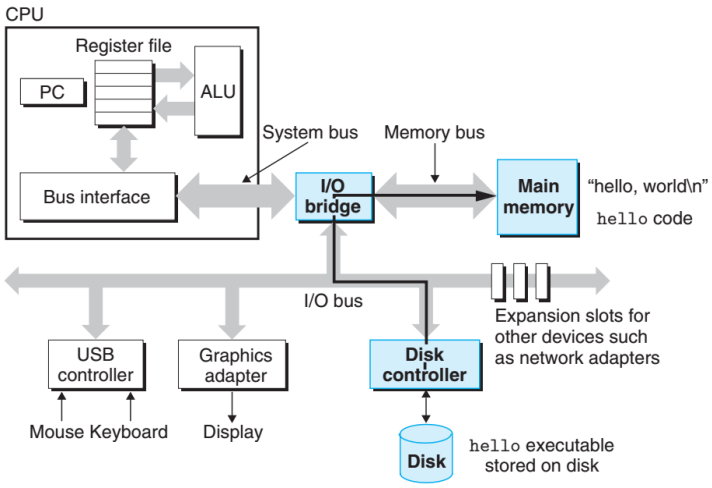


- When the `enter` key is pressed, the shell loads the executable `hello` file by executing a sequence of instructions that copies the code and data in the

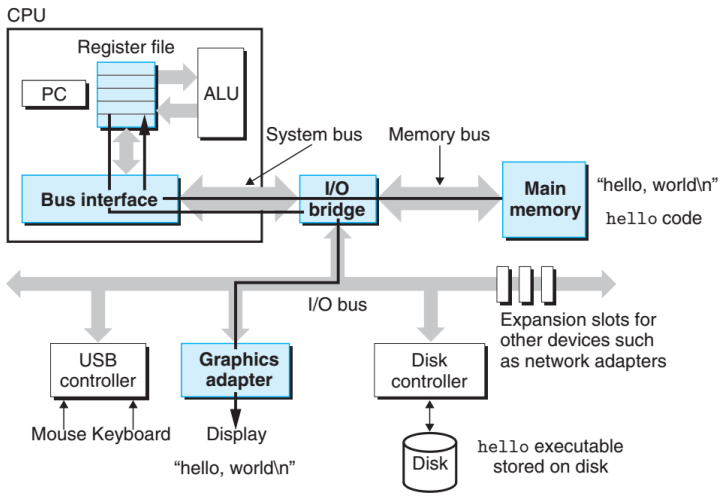
`hello` object file from disk to main memory.

  - The data include the string of characters "hello, world\n" that will eventually be printed out.

- Using a technique called *direct memory access* (DMA), the data travels directly from disk to main memory, without passing through the processor.

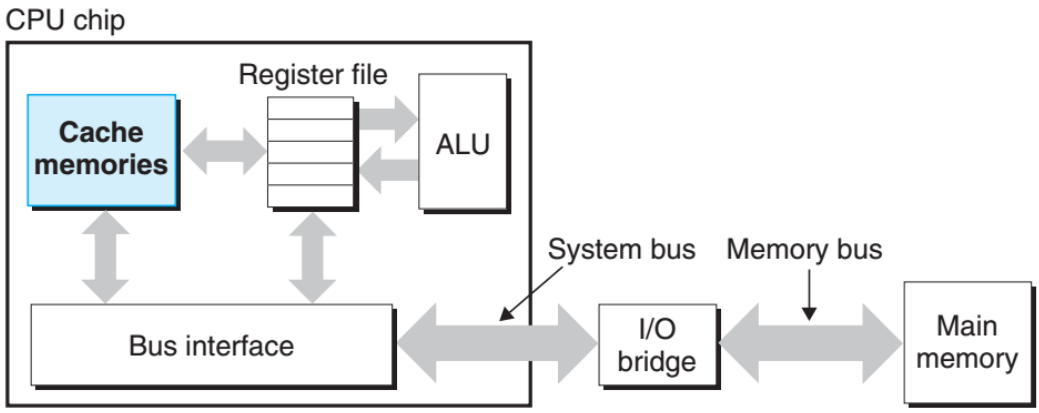


- Once the code and data in the `hello` object file are loaded into memory, the processor begins executing the machine-language instructions in the **hello** program's `main` routine.
- These instructions copy the bytes in the "hello, world\n" string from memory to the register file, and from there to the display device, where they are displayed on the screen.



Caches

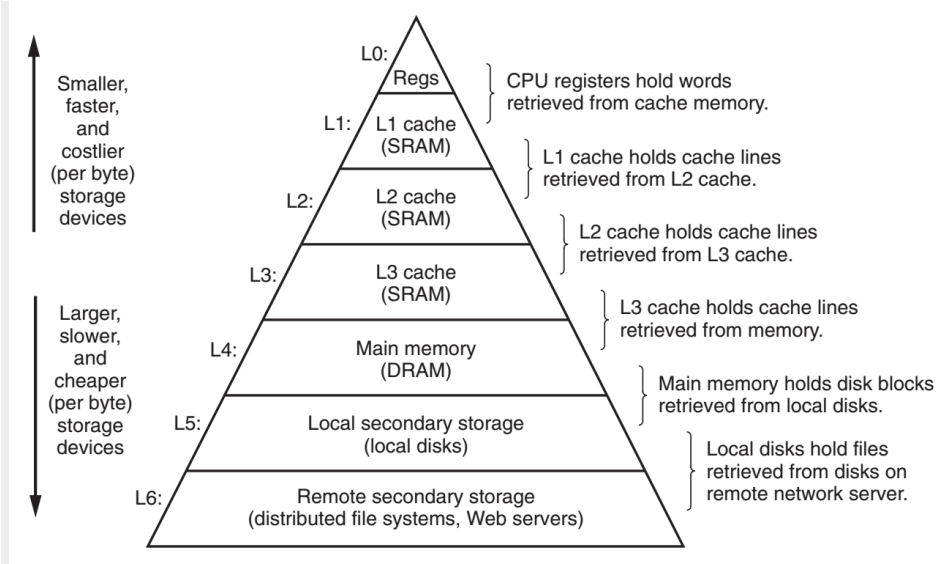
- Larger storage devices are slower than smaller storage devices.
  - For example — the disk drive on a typical system might be 1000 times larger than the main memory, but it might take the processor 10,1000,000 times longer to read a word from disk than from memory.
  - Similarly, a typical register file stores a few hundred bytes of information, as opposed to billions of bytes in the main memory. However, the processor can read data from the register file almost 100 times faster than from memory.
- It is easier and cheaper to make processors run faster than it is to make main memory run faster.
- To deal with the *processor-memory gap*, the system designers include smaller faster storage devices called *cache memories* (or caches) that serve as temporary staging areas for information that the processor is likely to need in the near future.



- An *L1 cache* on the processor chip holds tens of thousand of bytes and can be accessed nearly as fast as the register file.
- A larger *L2 cache* with hundreds of thousands of bytes is connected to the processor by a special bus.
  - It might take 5 times longer for the processor to access L2 cache than the L1 cache, but this is still much faster than accessing memory.

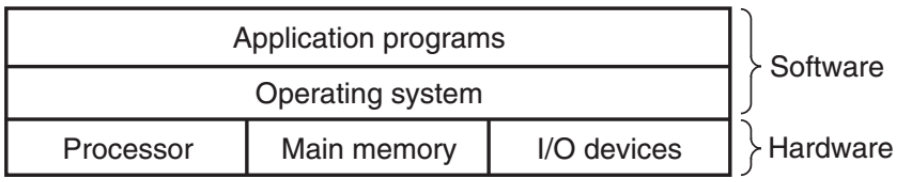
- The L1 and L2 caches are implemented with a hardware technology known as *static random access memory* (SRAM).

Storage devices form a Hierarchy



Operating system manages the hardware

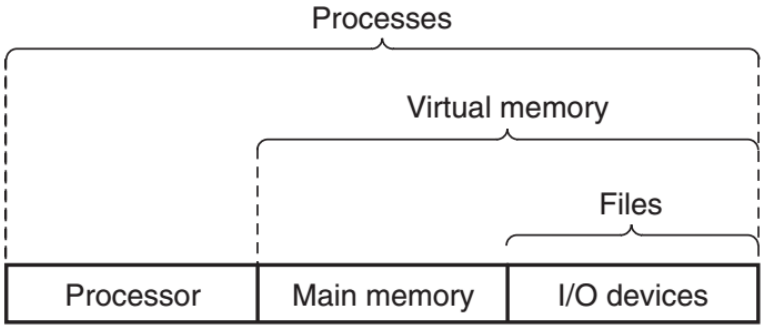
Operating system acts as a layer of software interposed between the application program and the hardware.



All attempts by an application program to manipulate the hardware go through the operating system.

- os protects the hardware from misuse by runaway applications
- os provides applications with simple and uniform mechanisms for manipulating complicated low-level hardware devices

The os achieves both goals via the fundamental abstractions: *processors, virtual memory, and files*.



Processes

When a program such as `hello` runs on a modern system, the os provides the illusion that the program is the only one running on the system.

A *process* is operating system's abstraction for a running program. Multiple processes can run concurrently on the same system, and each process appears to have exclusive use of the hardware.

A single CPU can appear to execute multiple processes concurrently by having the processor switch among them.

- The os performs this interleaving with a mechanism known as context switching.
    - It performs context switch by saving the context of the current process, restoring the context of the new process, and then passing control to the new process.
    - The new process picks up exactly where it left off.

Threads

In modern systems a process can actually consist of multiple execution unites, called *threads*, each running in the context of the process and sharing the same code and global data.

Threads are typically more efficient than processes.

Multi-threading is one way to make programs run faster when multiple processes are available

Virtual Memory

Virtual memory is an abstraction that provides each process with the illusion that it has exclusive use of the main memory.

Each process has the same uniform view of memory, which is known as its *virtual address space*.

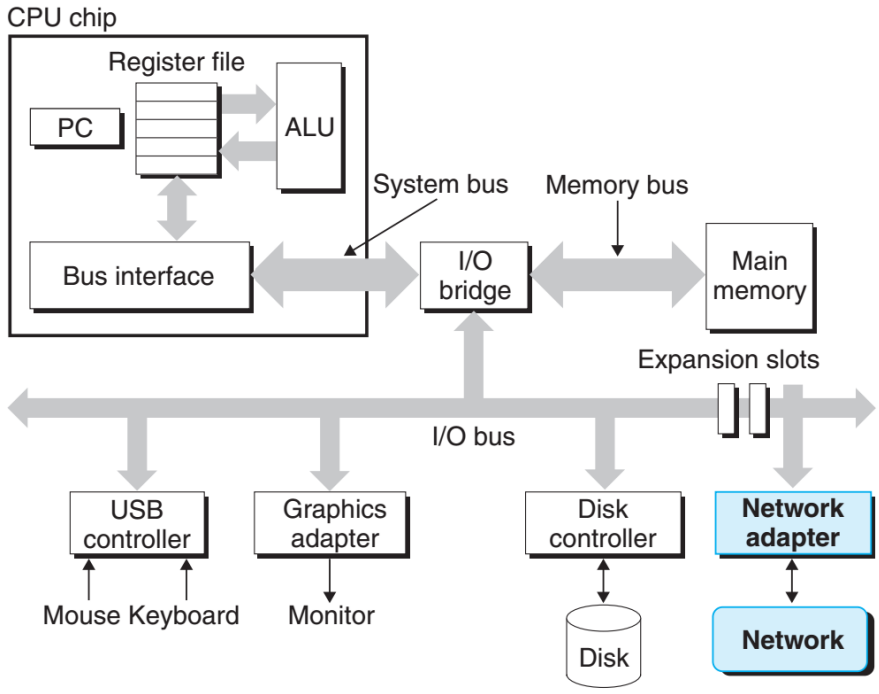
Files

A *file* is a sequence of bytes. Every I/O device, including disks, keyboards, displays, and even networks, is modeled as a file.

All input and output in the system is performed by reading and writing files, using a small set of system calls known as *Unix I/O*.

Communication between Systems (using networks)

In practice, modern systems are often linked to other systems by networks. From the point of an individual system the network can be viewed as just another I/O device.



When the system copies a sequence of bytes from main memory to the network adapter, the data flow across the network to another machine.

With the advent of global networks such as the Internet, copying information from one machine to another has become one of the most important uses of computer systems.

- For example — applications such as email, instant messaging, the World Wide Web, FTP, and telnet are all based on the ability to copy information over a network.

Important themes

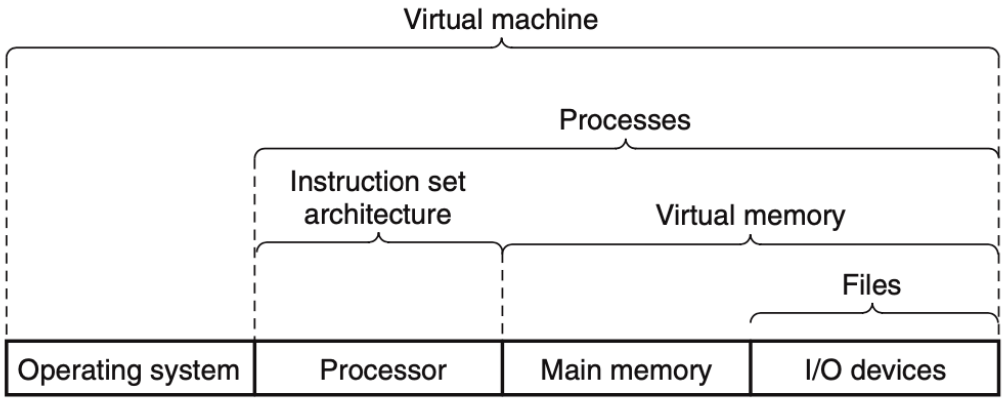
1. Concurrency and Parallelism

Concurrency is used to refer to the general concept of a system with multiple, simultaneous activities, and the term *parallelism* to refer to the use of concurrency to make a system run faster.

Parallelism can be exploited at multiple levels of abstraction.

2. Importance of abstractions in computer systems

The use of *abstractions* is very essential for computer architecture. Abstractions prov



- Instructional Set Architecture provides an abstraction of the actual processor hardware.
- On the operating system side — files as an abstraction of I/O, virtual memory as an abstraction of program memory, and processes as an abstraction of a running program.
- Virtual machine provides an abstraction for an entire computer, including the operating system, the processor, and the programs.

3. Amdahl's Law

The main idea is that when one part of the system is sped up, the effect on overall system performance depends on both how significant this part was and how much it sped up.

- Consider a system in which executing some application requires time T_{old}.
- Suppose some part of the system requires a fraction α of this time.
- Now the performance of this part of the system is improved by a factor of k.
    - This means that this component initially required time αT_{old} and now it requires $(\alpha T_{old})/k$.
- Therefore the new overall execution time would be:

$$\begin{aligned} T_{new} &= (1-\alpha)T_{old} + (\alpha T_{old})/k \\ &= T_{old}[(1-\alpha) + \alpha/k] \end{aligned}$$

- From this, The overall speedup $S = T_{old}/T_{new}$ is:

$$S = \frac{1}{(1-\alpha) + \alpha/k}$$